Reasoning about VHDL and VHDL-AMS using denotational semantics *

P.T. Breuer[1] N. Martinez Madrid[1] J.P. Bowen[2] R. France[3] M. Larrondo Petrie[3]
C. Delgado Kloos[1]

[1] Área de Ingeniería Telemática, Universidad Carlos III de Madrid
Butarque 15, E–28911 Leganés/Madrid, Spain.
{ptb,nmadrid,cdk}@it.uc3m.es

[2] Department of Computer Science, The University of Reading
Whiteknights PO Box 225, Reading, Berks RG6 6AY, UK
J.P.Bowen@reading.ac.uk

[3] Department of Computer Science and Engineering, Florida Atlantic University
777 Glades Road, Boca Raton, FL, USA 33431-0991
{robert,maria}@cse.fau.edu

Abstract

This paper introduces a denotational semantics for a core of the draft IEEE standard analog and mixed signal design language VHDL-AMS, and derives general results about the behaviour of VHDL-AMS programs from it. We include, for example, a demonstration that VHDL-AMS parallelism is benign in the absence of shared initializations. As proof of concept we have built an interpreted simulator that prototypes the semantics and which runs multi-process mixed analog and digital descriptions correctly.
Keywords*: Language design, VHDL-AMS, Language semantics, mixed-signal simulation.*

1 Introduction

VHDL-AMS seeks to *conservatively extend* the IEEE standard digital design language VHDL [7] to cover mixed analogue and digital systems. It adjoins linguistic elements that model continuously varying analogue systems. VHDL is intended to be conserved in the sense that, firstly, it forms a correct subset of the extended language syntax, and, secondly, code in the VHDL subset should run under VHDL-AMS in such a manner that, when viewed at integer unit time points, its trace is exactly that under VHDL. It would be useful to have formal assurances of this, else users cannot be certain that the results from new VHDL/VHDL-AMS simulators are valid for VHDL alone.

*This research has been partially funded by NATO grant CRG.960228 CARE4HW.

Proving anything about a real-world language, and impossible without a suitable basis for proof, so this article first sets out a denotational semantics for VHDL-AMS. We cover a core subset of the *elaborated* language (a subset in terms of which the rest of the language is defined). The subset includes scheduled signal assignments, wait statements, analogue differential equations for *quantities*, their re-initializations, and all standard imperative language control constructs.

The semantics presented here is one of two models for VHDL-AMS that we know of. The other has been developed by Sasaki et. al. [9] and extends a model of VHDL '93 given in terms of Evolving Algebras in the chapter by Börger et. al. in [6]. The present semantics is derived from the denotational semantics of Breuer et. al. for unit delay VHDL, as presented in [6] in functional style, later recast as relational semantics with a complete axiomatic semantics and refinement calculus in [1, 2, 3]. In [2], time was treated as continuous with integer unit observation points, which paves the way for the development here, sketched in [8], in which observations may also be continuous.

Although denotational style semantics have proved successful in the development of formal methods, no "theorems in the large" have been generated from them. That is in contrast to the situation in operational semantics, where Kees Goosens' operational semantics for VHDL'93 [5] was used by him to prove that the parallelism in VHDL was benign, in the sense that no ambiguity derives from it (Goosens was here talking about a subset that excluded simultaneous initializations of shared variables). In other words, VHDL simulation runs are intrinsically repeatable. In this article

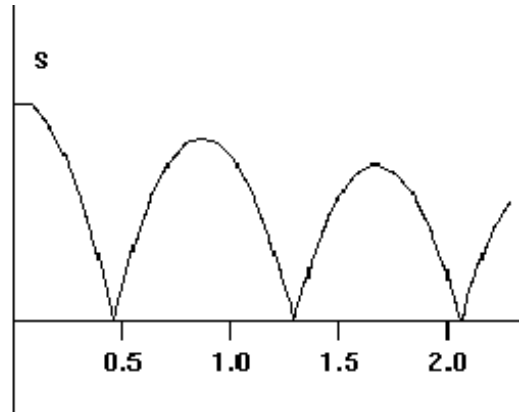

Figure 1. Bouncing ball (height vs. time)

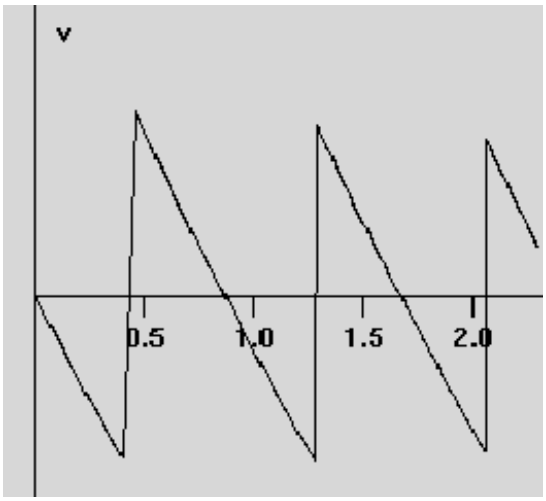

Figure 2. Bouncing ball (velocity vs. time)

we show that denotational semantics is capable of the same result, this time in the extended context of VHDL-AMS. We also show or sketch proofs that VHDL-AMS does indeed extend VHDL conservatively, and that VHDL-AMS has reasonable properties with respect to the numerical imprecisions, unavoidable in analogue systems simulation.

For example, we have studied a small mixed-signal single-process program drawn from the discussions over the VHDL-AMS draft that is intended to simulate a ball bouncing under the influence of gravity, and feeling some air resistance (Simulation in Figure 1 and Figure 2). In the core syntax in which we express VHDL-AMS, the program is:

```
process ball(qout: h=1, v=0; var g = 9.8, a = 0.1)
  d v/dt == -g - sign(v) * a * v^2;
  d h/dt == v;
begin
    wait until h < 0
    break h => 0;
    break v => -v;
end
```

Surprisingly, instead of a decaying bounce height, a computer simulation may easily produce an increasing bounce. The most interesting aspect is that if the computer is late in seeing the bounce, the ball will be traveling a faster than it would have been at the correct moment. Reversing the sign of the velocity (`break v => -v`) and translating it to its correct bounce position (`break h => 0`) has the effect of moving it up a potential energy gradient without decreasing the extra kinetic energy. That shows up as increased bounce next time round the loop.

The astute programmer will correct for possible excess energy:

```
process ball(qout: h=1, v=0; var g = 9.8, a = 0.1)
  d v/dt == -g - sign(v) * a * v^2;
  d h/dt == v;
begin
    wait until h < 0
    break h => 0;
    break v => - sqrt(2) * sqrt(v^2 / 2  + g * h);
end
```

But the programmer cannot always know or guard against this kind of problem. Instead he or she must hope that increasing numerical precision and sampling frequency leads to increasing overall accuracy. We can prove that that is the case under reasonable hypotheses, but that even then the assurance only holds for "all but a finite number of points of discontinuity" in any finite interval. What may happen, for example, is that the approximants to, say, a sawtooth waveform all err timewise by some positive epsilon. Epsilon shrinks to zero as the mesh used to solve the differential equations gets finer. But the error at the discontinuities in the sawtooth will always be of the order of the size of the wave. It is not trivial to prove even that much.

The approach required to prove this requires a second semantics for VHDL-AMS, this time based not on a continuous time line, but on a mesh of an arbitrary minimal precision. In this article we will sketch the apparatus required to study the relation between the "ideal" and the "approximation" semantics in the limit.

We would wish to also derive other guarantees of good behaviour. For example, that "reality is a fixpoint of the scheduling semantics": if we run a simulation once, and it gives rise to a certain sequence of events, then we can rerun the simulation, this time giving the last simulation results as an initial schedule of events, and the same sequence of events will ensue.

This observation is the basis for a prototype VHDL-AMS simulator that we have built. But we do not have as yet a denotational proof of the result. The simulator cal-

culates the schedule that is the joint 'reality' fixpoint of all processes. The Figures 1 and 2 are outputs from that simulator.

The layout of this paper is as follows. In Section 2 the simple core language syntax is set out. In Section 3 an "ideal" formal denotational semantics is given in terms of classical dynamic systems, and general observations on the behaviour of VHDL-AMS programs are derived from it. Section 4 comments briefly on the translation of this semantics into a prototype simulator implementation.

2 Core Syntax

This section describes a core syntax for VHDL-AMS, into which we translate other constructs of the language. It is intended to approximate an abstract syntax for standard *elaborated* VHDL-AMS (macros and non-primitive constructs expanded).

```
prog ::= p1 p2 p3 ...
proc ::= process n1(d1;d2;d3;...)
            e1; e2; ...
         begin
            s1; s2; ...
         end
decl ::= qout q1[=k1], q2[=k2], ...
       | qin  q1[=k1], q2[=k2], ...
       | out  g1[=k1], g2[=k2], ...
       | in   g1[=k1], g2[=k2], ...
       | var  v1[=k1], v2[=k2], ...
eqn  ::= d q1/dt == x1
expr ::= k1 | -x1 | x1+x2 | x1*x2 | ...
       | f1(x1,x2,...) | v1 | g1 | q1
       | if b1 then x2 else x3 fi
bool ::= T | F | b1 && b2 | x1 < x2 | ...
stmt ::= if b1 then s1;... else s2;... fi
       | while b1 do s1;s2;... od
       | do s1;s2;... until b1
       | null
       | wait [on x1] until b1
       | send g1 <= x1 [after x2]
       | break q1 => x1
       | v1 := x1
       | relax
```

$d_i \in$ `decl`, $e_i \in$ `eqn`, $q_i \in$ `qty`, $s_i \in$ `stmt`, $v_i \in$ `var`, $g_i \in$ `sgnl`, $b_i \in$ `bool`, $x_i \in$ `expr`, $k_i \in$ `cnst`, $f_i \in$ `func`, $p_i \in$ `proc`, $n_i \in$ `pid`, $i = 1, 2, 3, \ldots$.

Figure 3. Syntax of the core language

The language contains the following additional constructs over and above what would be required for VHDL, reflecting the extensions made in VHDL-AMS:

1. Processes may declare *quantities*, as well as signals and local variables.

2. *Differential equations* governing quantities may be declared.

3. *Expressions* may reference quantities as well as signals and variables.

4. The `break` statement reinitializes a quantity at the next entry into the differential equation solver.

5. The `wait` statement may wait upon a condition that references an analog quantity, as well as signals and variables. In the draft standard a process can wait upon a quantity crossing a threshold (`wait on` q`'above(`x`)`).

Resolution functions, which in VHDL are used to arbitrate the results of simultaneous writes to the same signal, are not allowed here. They may be simulated by replacing every reference to a resolved signal by the expression that its resolution function denotes.

The following aspects may be regarded as either additions or restrictions with respect to the VHDL-AMS standard. They chiefly provide for localization of the control of quantities in a single process, which formed a part of our experimentation.

1. *Export* (via `qout`) of a quantity is allowed in a process declaration.

2. The equations governing a quantity are *local* to the process that exports the quantity.

3. *Relax* is a new pseudo-statement.

4. Only differential equations of the simple form $\mathbf{d}v/\mathbf{d}t = \ldots$, in which the r.h.s. contains no differentials, are allowed. Theoretically, this is sufficient.

In the VHDL-AMS standard, quantities may only be globals. That approach may be duplicated here by encapsulating all quantities in one process.

Relax is the trivial case of *break.* It has the effect of reapplying the current differential equations in order to obtain new forward projections in the scheduler.

All quantities, variables and signals are real valued. The name space is shared.

3 Denotational Semantics

The semantics given to VHDL-AMS here is based on our published semantics for (unit delay) VHDL [1, 2]. The only essential changes to the domain equations are (1) in the underlying domain of time, which for VHDL is integer-based, and for VHDL-AMS is based on the real numbers

plus deltas, and (2) in the predication of the semantic functions on an *oracle*, the 'black box' charged with solving differential equations to give a state trajectory.

The *termination semantics* $\mathcal{S}[s] \in \texttt{SS}$ of a statement s is a (causal) relation between the system state before the execution of s, and after its termination.

The dynamic state of a system is captured in a *worldline, time point* pair. A *worldline* is a record of the historical, current, and projected states of the system at all possible times, and the *time point* identifies the state in the world line that is considered current. As a system evolves, the time point moves forward, and the projected future states change, in accordance with those VHDL scheduling commands that have been executed. The historical part of the world line never changes as the time point moves forward, reflecting causal behaviour.

Time is a two-dimensional pair t^i consisting of (1) the current real time $t \in \Re$ of the system and (2) the index of the current *simulation delta cycle* $i \in \mathcal{N}$. I.e time extends the reals, associating an infinite number of points $t^0 < t^1 < t^2 < \ldots$ to each real time $t \in \Re$. The extra points $t^1, t^2, \ldots$ represent repeated simulation cycles at the same instant of time, or *delta*s. Processes may engage in communication not only at the "ordinary" times t^0, but also at the delta times $t^1, t^2, \ldots$. Progress in time is represented either by an increment of the delta ordinate while real time stays constant, or an increase in the real time ordinate.

```
Time  = (ℜ,𝒩)
State = Id → ℜ
WL    = Time → State
SS    = (WL,TP)  ⟷(causal) (WL,TP)
SP    = (WL,TP)  ⟷(causal) (WL,TP)
Semantics = (SS,SP)
```

Figure 4. Semantic Domains

Whilst a termination semantics is appropriate for statements without duration, it is inadequate for constructs s that may never terminate, or terminate only after an indefinite time in which they do something observable. A VHDL-AMS *process* never terminates, but produces an observable trace. Therefore we define a *suspension semantics* $\mathcal{P}[s]$.

$\mathcal{P}[s]$ is the relation between a system at the onset of execution of s, and the system at any point at which the execution of s is still in progress. The latter may be thought of as the observable system state when the observer chooses to send a *suspend* signal to a running process.

The suspension semantics for statements that have no duration, such as assignment, is empty. The termination semantics for statements that do not terminate is the chaotic (causal, history preserving) relation.

The semantic functions are predicated on an oracle $\mathcal{O}$. This is an agent that predicts the trajectories of quantities. An oracle, given an initial state and time, and queried about a future time, will pronounce on the state then:

```
Oracle = (State,Time) → Time → State
```

Sequences of statements compose as sequences of relations. The loop fixpoint is the largest (i.e. least determined) relation that satisfies the causality (history preserving) constraint. Causality is necessary because otherwise the semantics of a statement followed by a nonterminating loop would be historically indeterminate, even though the preceeding statement had been well-behaved.

```
𝒮[-] :: stmt → Oracle → SS

𝒮[if b1 then ss1 else ss2 fi]𝒪(w,t) =
  if [b1](wt) then 𝒮[ss1]𝒪(w,t) else 𝒮[ss2]𝒪(w,t)
𝒮[while b1 do ss1 od]𝒪=
  𝒮[if b1 then ss1;while b1 do ss1 od else null]𝒪
𝒮[s1;...;sn]𝒪= 𝒮[s1]𝒪; ... ; 𝒮[sn]𝒪
𝒮[null]𝒪(w,t) = {(w,t)}
𝒮[do ss1 until b1]𝒪= 𝒮[ss1;while b1 do ss1 od]𝒪
```

Assignments here consist of a system state change followed by a *relax*. So long as at least one *relax* precedes the entry into the next wait statement, the wait will see the correct system projections. The laziest and simplest implementation is to place a prior *relax* as part of each wait statement semantics, but we also place *relax*'s immediately after assignments here in order to keep the denotation looking natural.

```
𝒮[send g1 <= x1 after x2]𝒪(w,t^i) = 𝒮[relax](w',t^i)
  where d = [x2](wt)
  w' = w⊕{t'↦wt'⊕{g1↦[x1](wt)}|t'≥t+d}, d>0
     = w⊕{t'↦wt'⊕{g1↦[x1](wt)}|t'>t},   d≤0
𝒮[send g1 <= x1]𝒪= 𝒮[send g1 <= x1 after 0]𝒪
𝒮[break q1 => x1]𝒪(w,t) =
  𝒮[send q1'break <= not(q1'break);relax]𝒪(w',t)
  where w' = w ⊕ {t'↦wt'⊕{q1↦[x1](wt)}|t'>t}
𝒮[v1 := x1]𝒪(w,t) = {(w',t)}
  where w' = w ⊕ {t'↦wt'⊕{v1↦[x1](wt)}|t'≥t}
𝒮[relax]𝒪(w,t) = {(w',t)}
  where w' = w ⊕ {t'↦𝒪(wt)t'|t'>t}
𝒮[wait until b1]𝒪=
  𝒮[do {wait on q1'break||... until b1} until b1]𝒪
𝒮[wait on x1 until b1]𝒪(w,t^i) = {(w,t')}
  where t'=min{τ>t^i|[b1](wτ)∨[x1](wτ)≠[x1](wt^i)}
```

Note that a `break` q_1 `=>` x_1 generates a pseudo-signal q_1'`break`. The effect of the break is not felt until the next delta cycle, as with a zero delay signal assignment. A positively (non-zero) delayed signal assignment will always be felt in the zero'th delta at the scheduled time, provided it has not been pre-empted. A variable assignment is felt immediately.

The *relax* statement projects forwards the trajectories of the quantities, according to the oracle. The oracle solves the differential equations.

A *wait* statement always takes at least one delta. It exits momentarily on receipt of a break signal and updates the projected states using *relax*, then reenters the wait. A subtlety is that the wait should leave unconstrained all non-local signals and quantities, and that is implemented through the oracle semantics. Parallelism is represented by intersection of relations, and thus implements synchronous concurrency – the juxtaposed processes simultaneously agree on the state of the whole system at all times.

The suspension semantics is empty for assignments, and nonempty for waits. Sequences follow the laws set out in [1, 2]. Two statements in sequence may either suspend in the first or the first may complete and then the suspension must occur in the second.

$\mathcal{P}$[-] :: stmt → Oracle → SP

$\mathcal{P}$[if b_1 then ss_1 else ss_2 fi]$\mathcal{O}$(w,t) =
  if [b_1](wt) *then* $\mathcal{P}$[ss_1]$\mathcal{O}$(w,t) *else* $\mathcal{P}$[ss_2]$\mathcal{O}$(w,t)
$\mathcal{P}$[while b_1 do ss_1 od]$\mathcal{O}$=
  $\mathcal{P}$[if b_1 then ss_1;while b_1 do ss_1 od else null]$\mathcal{O}$
$\mathcal{P}$[s_1;s_2]$\mathcal{O}$= $\mathcal{P}$[s_1]$\mathcal{O}$∪ $\mathcal{S}$[s_1]$\mathcal{O}$;$\mathcal{P}$[s_2]$\mathcal{O}$
$\mathcal{P}$[null]$\mathcal{O}$(w,t) = {}
$\mathcal{P}$[do ss_1 until b_1]$\mathcal{O}$= $\mathcal{P}$[ss_1;while b_1 do ss_1 od]$\mathcal{O}$
$\mathcal{P}$[v_1 := x_1]$\mathcal{O}$(w,t) = {}
$\mathcal{P}$[send g_1 <= x_1 after x_2]$\mathcal{O}$(w,t^i) = {}
$\mathcal{P}$[send g_1 <= x_1]$\mathcal{O}$= {}
$\mathcal{P}$[break q_1 => x_1]$\mathcal{O}$(w,t) = {}
$\mathcal{P}$[relax]$\mathcal{O}$(w,t) = {}
$\mathcal{P}$[wait until b_1]$\mathcal{O}$=
  $\mathcal{P}$[do wait on q_1'break||... until b_1 until b_1]$\mathcal{O}$
$\mathcal{P}$[wait on x_1 until b_1]$\mathcal{O}$(w,t^i) = {(w,τ)|t^i≤τ<t'}
  where t'= min{τ>t^i|[b_1](wτ)∨[x_1](wτ)≠[x_1](wt^i)}

The suspension semantics defines the externally observable behaviour. The termination semantics defines internal changes of state. Assignments are not observable in themselves, but their indirect effects are observable later during a following wait statement.

Now to the semantics of processes: only the suspension semantics is of interest. A process is a loop, and parallelism is expressed by intersection of relations.

$\mathcal{P}$[-] :: proc → Oracle → SP
$\mathcal{P}$[process n_1(d_1;...) e_1;... begin s_1;... end]$\mathcal{O}$=
$\mathcal{S}$[d_1;...;e_1;...]$\mathcal{O}$';$\mathcal{P}$[while T do s_1;... od]$\mathcal{O}$'
  where $\mathcal{O}$'=D[e_1;...]$\mathcal{O}$

$\mathcal{P}$[-] :: prog → Oracle → SP
$\mathcal{P}$[p_1 ... p_n]$\mathcal{O}$= $\mathcal{P}$[p_1]$\mathcal{O}$∩ ... ∩ $\mathcal{P}$[p_n]$\mathcal{O}$

The oracle is built from differential equations.

D[-] :: (eqn;...;eqn) → Oracle → Oracle
D[d q_1/dt == x_1;...; d q_n/dt == x_n]$\mathcal{O}$= $\mathcal{O}$'
where $\mathcal{O}$'(s,t)t' = *if* t'>t *then* s' *else* s
      where d s'q_1/dt' = [x_1]s'
            ...
            d s'q_n/dt' = [x_n]s'
              s'q_m = $\mathcal{O}$(s,t)t'q_m, q_m≠q_1,...,q_n

Solving the equations for s' requires numerical integration. It has to be carried out using inputs from its environment that may involve other differential equations in a mutual recursion. To avoid infinite loops, each numerical calculation must delay its outputs fractionally with respect to its inputs.

Initializations set the value at zero time only, if they come from input declarations, or schedule the value for all time, if they come from output declarations.

$\mathcal{S}$[qout q_1=k_1]$\mathcal{O}$(w,t)={(w⊕{τ↦wτ⊕{q_1↦k_1}|τ ≥t},t)}
$\mathcal{S}$[qout q_1]$\mathcal{O}$= $\mathcal{S}$[qout q_1=0]$\mathcal{O}$
$\mathcal{S}$[qin q_1=k_1]$\mathcal{O}$(w,t)={(w⊕{t↦wt⊕{q_1↦k_1}},t)}
$\mathcal{S}$[qin q_1]$\mathcal{O}$= $\mathcal{S}$[qin q_1=0]$\mathcal{O}$
$\mathcal{S}$[out g_1=k_1]$\mathcal{O}$(w,t)={(w⊕{τ↦wτ⊕{g_1↦k_1}|τ ≥t},t)}
$\mathcal{S}$[out g_1]$\mathcal{O}$= $\mathcal{S}$[out g_1=0]$\mathcal{O}$
$\mathcal{S}$[in g_1=k_1]$\mathcal{O}$(w,t)={(w⊕{t↦wt⊕{g_1↦k_1}},t)}
$\mathcal{S}$[in g_1]$\mathcal{O}$= $\mathcal{S}$[in g_1=0]$\mathcal{O}$
$\mathcal{S}$[var v_1=k_1]$\mathcal{O}$(w,t)={(w⊕{τ↦wτ⊕{v_1↦k_1}|τ ≥t},t)}
$\mathcal{S}$[var v_1]$\mathcal{O}$= $\mathcal{S}$[var v_1=0]$\mathcal{O}$

We have omitted the interpretation of expressions and booleans. They interpret in the natural way against state, but there is no intrinsic reason why they should not interpret against a world line. That would allow the use of temporal logic with scheduling semantics in conditionals, giving rise to an interesting extension of VHDL-AMS.

Note that the semantics is clearly conservative with respect to VHDL. In the absence of differential equations and quantities, the semantic functions reduce to the forms correct for VHDL, albeit over a larger time domain. Since all but the wait semantics are given by universally quantified predicates, only a little argument and some extra hypotheses are required in order to force a VHDL trace to satisfy the semantic equations for VHDL-AMS:

Proposition 1 *In the absence of quantities, and if the time variable is always accessed via truncation, and all external signals only change at integer times, then a VHDL-AMS program expressible in the core language behaves as though it were running in VHDL, when sampled at integer times.*

We need to show that a VHDL trace satisfies the VHDL-AMS semantic constraints because then determinism implies that it is the VHDL-AMS trace too. The argument goes: imagine that the VHDL-AMS wait semantics is modified to include a quantification over only integer/delta time points, so that it has the VHDL semantics. In principle, the time minimization "the first $\tau > t^i$ such that e" for some expression e may now give a bigger result. But e has time accesses guarded via truncation and all the signals change at only integer times, so if the minimization over $\Re$ gives a future time value t'^j with $t' > t$, then it will be (1) with $j = 0$ since signal changes between strictly future deltas cannot be scheduled and quantities are not projected to change during

deltas and (2) with t' integer. So t' minimizing e is the same integer time value whether quantified over integers or reals. On the other hand, if the quantification gives the same major time, i.e. $t^0 = t_0$, then the wait is exactly until the next delta cycle, as in VHDL. So the VHDL-AMS behaviour satisfies VHDL semantic equations. □

The kind of statement that would have differing semantics, had not care been taken to exclude it, is

```
wait until t*2 >= 1
```

which might wait until `t = 0.5`, but the only equivalent allowed here is

```
wait until trunc(t)*2 >= 1
```

which will wait until `t = 1` if `t < 1`, in both VHDL-AMS and VHDL, and otherwise until the next delta cycle, in both. I.e. VHDL-AMS programs must be expressible in the core language as programs of the second form if VHDL programs are to have VHDL semantics in VHDL-AMS. The text of the draft standard is not very clear in this area. Rounding the time accesses is not sufficient to preserve the VHDL behaviour. C.f. the above with rounding in place of truncation.

Giving VHDL-AMS an integer-time semantics is at the heart of the argument above. Giving VHDL-AMS a *mesh-based semantics* is what is required in order to argue about its behaviour with respect to calculation errors. The semantics requires only a further change of time domain:

```
Time = (Zδ,N)
```

in which $\mathcal{Z}\delta$ is a set of at most δ-separated mesh-points in $\Re$. The same semantic relations hold as given earlier, with quantization now over $\mathcal{Z}\delta$ where appropriate. Exits from a wait statement are now at the next grid point after the correct exit time. We will not prove it here but it is true that:

Proposition 2 *If, over any bounded interval, the solutions to the differential equations depend uniformly on the inputs over that intervals, then: as a time-wise computation mesh $\mathcal{Z}\delta$ gets uniformly finer, the behaviour of a program executed over the mesh tends uniformly to its ideal behaviour in VHDL-AMS over every closed interval of real time in which it does not loop forever, except possibly those that contain an exit or entry from/to a wait statement in the VHDL-AMS ideal semantics.*

The proof depends on the argument that the control flow in a program eventually stabilizes as the mesh gets finer. If the computation makes progress forever, then it can only examine its inputs to a certain precision in every closed interval. Once the mesh gets finer than that the control flow in the program is fixed over that interval. A finer mesh just refines the calculations in each subinterval that the process spends in a wait statement.

We will sketch, however:

Proposition 3 *The parallelism in the core language is* benign *(and so is that in VHDL-AMS, to the extent that it is supported by the core), in the sense that the results of a simulator run never vary. I.e., every observable value is either completely undefined (capable of taking every possible value) or uniquely defined.*

The proof goes as follows: abstract the base domain (the real numbers $\Re$) to the domain $\underline{3} = \{\top > r > \bot\}$, in which $\top$ represents "no feasible result" and $\bot$ represents "all possible values," and $r \in \Re$ represents "unique value." The tables for basic operations such as addition are as follows:

$+$	$\top$	r_2	$\bot$
$\top$	$\top$	$\top$	$\top$
r_1	$\top$	$r_1 + r_2$	$\bot$
$\bot$	$\top$	$\bot$	$\bot$

The table is monotonic – the results get less refined from top to bottom and left to right. The plan of the proof is to (1) perform an *abstract interpretation* [4] of the semantic relations over this domain, (2) note that the semantic relations of individual processes become (monotonic) semantic functions over this domain. (3) note that the interpretation is exact for determinate input states. I.e. if in the interpreted domain the semantic function f gives a state w' with value $\bot$ at time t' for some variables x, then it is because the underlying semantic relation R relates the input state wt to all possible values of those variables x, independently:

$$fwtt'x = \bot,\ w't'|_{\overline{\{x\}}} = w''t'|_{\overline{\{x\}}} \rightarrow wtRw't' = wtRw''t'$$

This property is preserved through sequential composition of relations (equivalently, composition of functions in the interpreted domain). Therefore, (4) during a simulation run on one thread, a state variable is either fully determined or fully indeterminate at any moment. When finally we put the processes in parallel threads together to get a live thread (one without states that are in forced disagreement), all state variables at a given time are either (a) determined and take the same value in both threads, (b) determined in one and undetermined in the other, or (c) undetermined in both. The latter are the only surviving sources of nondeterminism in the parallel association, and they arise from the state variables fully undetermined in all threads, i.e. those that are not set in any thread.

The interesting part is rereading the nondeterministic semantic relations $R :: (\mathtt{WL}, \mathtt{Time}) \leftrightarrow (\mathtt{WL}, \mathtt{Time})$ as deterministic functions $f :: \mathtt{WL}[\underline{3}] \rightarrow \mathtt{Time} \rightarrow \mathtt{WL}[\underline{3}]$ (ignore the calculation of the new time point for clarity), where $\mathtt{WL}[\underline{3}] = \mathtt{Time} \rightarrow \mathtt{State}[\underline{3}]$ and $\mathtt{State}[\underline{3}] = \mathtt{Id} \rightarrow \underline{3}$.

We say that $fwtt'x = r$ if there is exactly one r such that $x = r$ holds in the w' with $wtRw't'$. If there is more than one we say that $fwtt'x = \bot$. If there is none then $fwtt'x = \bot$.

Now the property for step (3) has to be checked for the atomic relations and their interpretations. It trivially holds, for example, of assignment, because the semantics given here is a function. It is the `relax` semantics that introduces nondeterminism in the variables outside the control of the current execution thread, and it introduces full nondeterminism in those. □

The reasoning above highlights that VHDL-AMS semantics could be given in a more nondeterministic style. We have chosen instead to build all the nondeterminism into appeals to the oracle solving the differential equations. The rest of the process semantics here expresses deterministic dependence on what was returned by the oracle. But it would also be valid to build explicit variation in the uncontrolled variables into every semantic function within a process.

4 A Simulator

The semantics given above has been implemented with only minor modifications, chiefly to the parallelism semantics.

The program consists of 2000 code-lines in the functional programming language *Gofer*, and its extension *TkGofer*. *TkGofer* has a built-in interface to `tk`/`tcl` and generates displays under the X windows system.

The modifications eliminate the (benign) nondeterminism in parallelism. Each process is converted into a worldline-to-worldline transformer by regarding the initial schedule as input and the final trace as output. We know that the final trace is a fixpoint, when regarded as a schedule (not proved in this article), and so we solve for it as a fixpoint jointly produced by a group of processes, and jointly consumed by them. Great care has to be taken not to introduce extra strictness inadvertently. For efficiency, world lines are modelled as infinite lists (streams) rather than as real valued functions. To avoid mutual recursion between two of our "black box process semantics" for the number of computational deltas that have to be performed until stability, we set a fixed maximum for each simulation run. In the real world of VHDL-AMS simulators, the simulation kernel effectively peeks into each process to see whether there are further changes pending, but our compositional approach precludes that.

We used an interesting real-valued interpretation of logic in order to apply an efficient Newton-Raphson interpolation for the timepoints at which boolean conditions flip, rather than have to rely on numerical bisection.

5 Conclusion

A denotational semantics for the core of VHDL-AMS has been set out. It has been shown that parallelism in the core is benign, and it has been observed that the semantics is the unique limit of time-mesh based approximations. A proof that the semantics conservatively extends the VHDL semantics has been sketched.

References

[1] P.T. Breuer, N. Martínez Madrid, L. Sánchez, A. Marín, and C. Delgado Kloos. A formal method for specification and refinement of real-time systems. In *Proc. 8'th EuroMicro Workshop on Real Time Systems*, pages 34–42. IEEE Press, July 1996. L'Aquilla, Italy.

[2] P.T. Breuer, C. Delgado Kloos, N. Martínez Madrid, A. López Marin, L. Sánchez. A Refinement Calculus for the Synthesis of Verified Hardware Descriptions in VHDL. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **19(4)**:586–616, July 1997.

[3] P.T. Breuer, L. Sánchez, and C. Delgado Kloos. A simple denotational semantics, proof theory and a validation condition generator for VHDL. *Formal Methods for System Design*, **7(1 & 2)**:27–51, December 1995.

[4] P. Cousot and R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, In *Proc. 4th ACM Symposium on the Principles of Programming Languages*, pages 238–252, 1977.

[5] K.G.W. Goosens. Reasoning about VHDL using operational and observational semantics. In *Proc. Advanced Research Worksop on Correct Hardware Design Methodologies*, ESPRIT CHARME, Springer Verlag, October 1995.

[6] C. Delgado Kloos and P.T. Breuer *Formal Semantics for VHDL*. Kluwer Press, 1996.

[7] IEEE. *IEEE Standard VHDL Language Reference Manual, ANSI/IEEE STD 1076-1993*. Institute of Electrical and Electronic Engineers, New York, 1994.

[8] N. Martínez Madrid, P.T. Breuer and C. Delgado Kloos. A Semantic Model for VHDL-AMS. In *Proc. Conference on Correct Hardware Design and Verification Methods, CHARME'97*, October 16–18, 1997, Montreal, Canada, pages 106–126. IFIP; Chapman and Hall, 1997.

[9] H. Sasaki, K. Mizushima and T. Sasaki. Semantic Validation of VHDL-AMS by an Abstract State Machine. In *Proc. BMAS'97 (IEEE/VIUF International Workshop on Behavioral Modeling and Simulation)*, pages 61–68, October 20–21, 1997. Arlington, VA, USA.